Received: 26 February 2021 | Revised: 13 October 2021 | Accepted: 1 November 2021

DOI: 10.1002/cpe.6771

SPECIAL ISSUE PAPER

WILEY

SimAnMo—A parallelized runtime model generator

Michael Burger | **Giang Nam Nguyen** | **Christian Bischof**

Scientific Computing, Technical University of Darmstadt, Darmstadt, Germany

Correspondence
Giang Nam Nguyen, Scientific Computing, Technical University of Darmstadt, Alexanderstrasse 2, 64283 Darmstadt, Hesse, Germany.
Email: giang_nam.nguyen@tu-darmstadt.de

Funding information
Deutsche Forschungsgemeinschaft, Grant/Award Number: SFB 1119 236615297

Abstract

In this article, we present the novel features of the recent version of SimAnMo, the Simulated Annealing Modeler. The tool creates models that correlate the size of one input parameter of an application to the corresponding runtime and thus SimAnMo allows predictions for larger input sizes. A focus lies on applications whose runtime grows exponentially in the input parameter size. Such programs are, for example, of high interest for cryptanalysis to analyze practical security of traditional and post-quantum secure schemes. However, SimAnMo also generates reliable models for the widespread case of polynomial runtime behavior and also for the important case of factorial runtime increase. SimAnMo's model generation is based on a parallelized simulated annealing procedure and heuristically minimizes the costs of a model. Those may rely on different quality metrics. Insights into SimAnMo's software design and its usage are provided. We demonstrate the quality of SimAnMo's models for different algorithms from various application fields. We show that our approach also works well on ARM architectures.

KEYWORDS

exponential runtime, factorial runtime, runtime modeling, runtime prediction

1 | INTRODUCTION AND MOTIVATION

In cryptanalysis, one determines the hardness of the mathematical problems which underlie cryptographic schemes. To that end, one tries to efficiently solve those problems on modern HPC systems with highly parallelized attack algorithms.[1-3] By definition, these are hard problems. That means that solving them for bigger problem sizes comes with an exponential, or at least very rapid, increase in the time to solution. However, knowledge of the expected runtimes is required to choose problem instances which result in secure but efficient cryptographic schemes. Hence, the creation of reliable models to estimate the runtime for problem sizes that cannot be solved in the foreseeable future is required.

In those programs the runtime often increases exponentially in the size of one input parameter p. They are called programs with *exponential runtime behavior* in the following. We limit our considerations to the case of a single parameter p.

But detailed knowledge of the expected runtime is also important in many other fields. For example, when allocating an appropriate number of resources and runtime slots on HPC systems for large simulation codes. In those simulation applications, the runtime normally increases polynomially (or in a combination of polynomials and logarithmic parts). They are called programs with *polynomial runtime behavior*.

A third important type is the *factorial runtime behavior*. There, the dominating term in the model is $p!$ which occur, for example, when solving combinatorial problems. In that case, often heuristics are applied to find an acceptable solution in reasonable time. In many cases, programs with exponential or factorial runtime contain sub-algorithms with different behavior like polynomial or with the special case of

constant runtime. Detailed knowledge of those sub-algorithm's cost development increases the understanding of the behavior of the overall program.

We call a value of the input parameter p a *feasible choice* if that value allows us to solve the problem in an acceptable average runtime and with the hardware resources of our test systems.

In this article, we demonstrate SimAnMo's applicability for all mentioned types of runtime behavior. For programs with exponential runtime behavior, we focus on algorithms from the cryptanalysis context. Exponential runtime behavior often occurs in this context (cf. Section 5.3). The use-cases for polynomial and factorial runtime are well-known basic problems. For example, dense linear algebra or combinatorial problems, also including a parallel, heuristic algorithm (cf. Sections 5.2 and 5.4).

1.1 | Linear-logarithmic approach for exponential runtime behavior

A common approach to create models for the runtime of programs with exponential runtime behavior is the following: (1) perform timing measurements for feasible choices of the input parameter p. (2) Logarithmize the timing values; and (3) perform a linear regression, resulting in a straight line l. It is described by $l(p) = c_0 + c_1 \cdot p$ with $c_0, c_1 \in \mathbb{R}$ being determined by the regression. We call this procedure the *linear-logarithmic* (lin-log) approach. It is, for example, employed in Ducas[4(fig.3)] where single lines are fitted to different measurement series of four implementations of the SubSieve algorithm and the implementation of enumeration with extreme pruning in the fplll library. Albrecht et al.[5(fig.3a)] fit two lin-log curves for the runtime measurements of their General Sieve Kernel (G6K) implementation. In Reference 6 (fig. 13), Aono et al. create a lin-log model for the data points generated by their enumeration cost-estimator. They compare those estimates for the runtime to timing results reported in the literature. All cited examples come from the research field of finding solutions to a specific lattice-based problem, called the shortest vector problem (SVP). We discuss the SVP in more detail in Section 5.3.0.5. The SVP and problems that can be efficiently transformed to an SVP, like the closest vector problem (CVP) or the learning with errors problem (LWE), are employed as the base problem of several post-quantum cryptographic systems.[7,8]

Theoretically, one could estimate the runtime for higher values of p, if one evaluates $2^{c_0+c_1 \cdot p}$. But Ducas[4] and Albrecht et al.[5] state not to employ extrapolations of their data for predictions of higher order. However, we think that it is important to get insights into the behavior of the most advance algorithms to understand the practical hardness of the mathematical problems. For example, Reference 6 (fig. 13) demonstrates that different theoretical estimates for the runtime of a specific problem may differ by many orders of magnitude.

1.2 | Motivating example

We start with an illustrative example to highlight the properties of programs with exponential runtime behavior, the drawbacks of the lin-log approach and the quality of the models we create with SimAnMo.

The runtime of the Schnorr-Euchner enumeration algorithm (cf. Section 5.3.0.6) is expected to grow in $2^{\mathcal{O}(n^2)}$ with $n$being the dimension of the enumerated lattice.[9]

As an example, we assume that the dependence of the runtime of a program *ENUM*implementing the enumeration and its input parameter n follows the function:

$$g_{\text{ENUM}}(n) = 0.4 + 10^{-3} \cdot 2^{0.01 \cdot n^2} \tag{1}$$

The offset 0.4 emulates a constant initialization time for the program and the coefficients 10^{-3} and 0.01 are chosen arbitrarily. The only prerequisite for them is that the result values of $g_{\text{ENUM}}(n)$ lie in a range that is representable by double precision numbers for the values of n considered. We evaluate $g_{\text{ENUM}}(n)$ for $n \in \{20, 21, 22, 23, 24, 25\}$ since in this region the resulting theoretical runtime is still short but the effect of the constant 0.4 does not dominate the actual behavior. The resulting values are represented by the diamonds in Figure 1.

Afterward, we create a model with the lin-log approach by linear regression, shown in orange. Our alternative model created by SimAnMo is shown in blue.

The comparison in Figure 1 shows that SimAnMo's model nearly coincides with the diamonds. In contrast, the lin-log model lies between the points since it lacks the flexibility to fit a curve. This mainly results from the fixed order of 1 for n in the exponent $-2.05 + 0.038 \cdot n^{\mathbf{1}}$, a natural limitation of the lin-log approach.

In the next step, we extrapolate the model function to higher values of n and compare them to the values resulting from evaluating Equation (1) at the respective n-values. The results are visualized in Figure 2. For the y-axis, we employ a logarithmic scale.

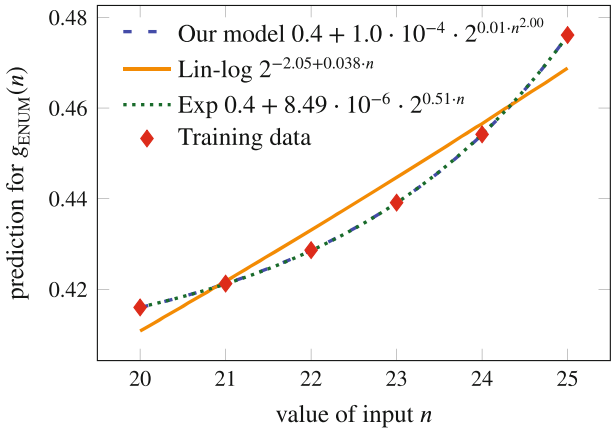


FIGURE 1 Models for $g_{\text{ENUM}}(n)$ created from the training points resulting from Equation (1)

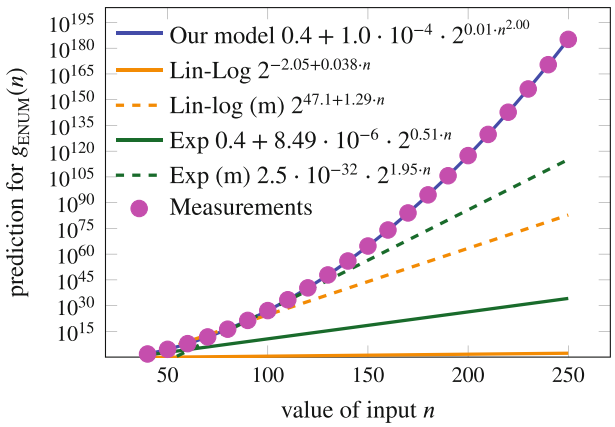


FIGURE 2 Prediction and evaluation for $g_{\text{ENUM}}(n)$-models

We see a huge difference between the models considered. The `SimAnMo` model fits the additional points quite exactly, while the lin-log model underestimates the values by more than 200 orders of magnitude. The coefficients in the diagram are rounded to two significant digits. The difference between our model and the real data point is smaller than 7% for $n = 250$.

A second lin-log model with suffix "(m)" is shown dashed. It is based on training points for $20 \leq n \leq 100$ in steps of 5. Although the model performs better, it underestimates the real value at $n = 250$ by more than 100 orders of magnitude. Again, the fixed order of n prevents an acceptable fit to the extended training data, resulting in the wrong prediction.

In addition, we also assess the quality of other models labeled as Exp and Exp (m). They are so called *exponential models* of the form $f(p) = c_0 + c_1 \cdot 2^{k \cdot p}$, $c_0, c_1, k \in \mathbb{R}$ and based on the same training data. We discuss them in Section 3.1.

This example suggests the accuracy of our models and highlights the lack of flexibility of the linear-logarithmic approach for a real algorithm.

Our contributions are:

1. We propose a new type of modeling functions for exponential runtime behavior
2. We demonstrate `SimAnMo`'s extensibility and flexibility by integrating and evaluating also factorial runtime models
3. We show that the resulting prediction of the models is more accurate when the target is to minimize the mean absolute percentage error (MAPE) instead of standard residual sum of squares (*RSS*).
4. We highlight the ease-of-use of `SimAnMo`

2 | RELATED WORK

Performance modeling is mainly employed to analyze and improve the performance of an application on a specific hardware platform. Besides analytical models, the most famous approach is the Roofline performance model[10] as a combination of the real system behavior (memory bandwidth) and knowledge of the code or the instructions it generates on a machine (number of FLOPs related to the number of memory accesses, i.e.,

computational intensity). It gives an upper bound for the reachable performance on a given architecture. The Roofline model was extended to take the cache hierarchy into account, for example, by Ilic et al.[11] Those cache-aware models were successfully applied to applications on large single compute nodes[12] or GPGPU.[13] Another approach for Roofline models on GPUs was presented by Ding and Williams.[14] Their Instruction Roofline methodology takes the various cache hierarchies of NVIDIA GPUs into account and considers more metrics than the computational intensity like strided memory accesses or instruction throughput. Two other extensions to the Roofline model were proposed by Lorenzo et al.[15] which include considering multiple measurements during different stages in the program execution and include the memory latency into the model.

A different approach is empirical performance model generation. Those models can, for example, be based on performing benchmarks with the target applications and employing machine learning[16] to generate the models. Another method is based on small-scale experiments/runs and to automatically extrapolate models from the results obtained. These techniques are applied in the tool Extra-P[17,18] which was created as a tool to assist developers of parallel programs in identifying scalability bugs by creating performance models of critical regions of the code. This is, in particular, important for the optimal usage of HPC systems. The approach is hardware agnostic in the sense that the modeler does not require knowledge about the system parameters like memory bandwidth, the configuration of the cache hierarchy or the GPU architecture employed. Also no detailed knowledge about properties of the code like the number of memory writes in specific parts is required.

In this article, our goal is not to analyze or improve the performance of applications by performance models on a given system but to generate models which estimate the runtime of a given algorithm with exponential runtime behavior for problem sizes which are infeasible to solve. It was already demonstrated that parts of the methodology of Extra-P can be employed to generate reliable models for the runtime of programs with polynomial runtime behavior[19] and that those models can be generated with low effort.

3 | MODEL GENERATION METHODOLOGY

In this chapter, we first present the approach to automatically generate runtime models from training data. It is followed by the definition of different metrics to evaluate the quality of models. Afterward, we present all types of models integrated into `SimAnMo` and finally discuss `SimAnMo`'s software design.

3.1 | Model generation

`SimAnMo` is based on principles of Extra-P.[20] There, functions are generated from profiled runtime data. This describes the relation of the runtime of different code regions with different variables, for example, the size of the input data. All models of Extra-P are of the form shown in Equation (2):

$$c_0 + \sum_{q=1}^{\nu} c_q \cdot p^{i_q} \cdot \log_2^{j_q}(p) \tag{2}$$

where p is the input parameter whose influence is considered and c_0 a constant term. Reisert et al.[21] have shown that in most cases $\nu = 1$ is sufficient to achieve a good result. That means the model looks like:

$$c_0 + c_1 \cdot p^i \cdot \log_2^j(p) \tag{3}$$

and, in particular, that those single term models are even the most accurate ones. When $c_0 = 0$ and $c_1 = 1$, the remaining part of the model is called a *constant model*. Its value depends on the other coefficients i and $j$as well as the value of p.

Appropriate values for the parameters c_0, c_1, i, and $j \in \mathbb{R}$ are determined in a two step approach. The exponents i and j are set by iterative refinement. After the choice of i and j, the two values c_0 and c_1 result from setting their values in order to minimize a given quality metric for already fixed exponents i and j. This means c_0 and c_1 are set in order to optimize the fit of the *constant models* to the training data points.

An experimental version of Extra-P also allows to create models of programs with exponential runtime behavior. The shape of those model functions is:

$$f(p) = c_0 + c_1 \cdot 2^{k \cdot p} \tag{4}$$

and we call those exponential models from our extended Extra-P version *exp models* in the following.

Independent of the model function employed, the workflow to create a model is the same. In the first step, so called *training data* is generated. It is represented by profiled runtime measurements with l varying p-values. In the second step, Extra-P's model generator determines the best models

that describe the training data points by regression, cross-validation, and iterative refinement (cf. Calotoiu et al.[20,21] for details). The regression tries to minimize the residual sum of squares (*RSS*) between the training data points and the model function which is defined as:

$$RSS = \sum_{m=1}^{l} (y_m - f_m)^2 \tag{5}$$

where m is the number of the measurement y_m and f_m is the evaluation of the model function f for the p-value corresponding to the mth measurement. The *RSS* is the standard least squares residual to evaluate the quality of a model. If $RSS = 0$, then $y_m = f_m \ \forall m$ and each point of the training data lies directly on the fitting curve.

3.2 | Quality assessment

`SimAnMo` is also based on collecting training data points and then creating a model function which minimizes a given cost function. Since one drawback of *RSS* is that the errors are summed up, that is, adding more measurements always increases the *RSS* value, `SimAnMo` supports several different quality metrics to analyze the quality of fit with the training data.

For our own metric *anRSS*, we start from the absolute normalized RSS (*nRSS*). This was presented by Ilyas et al.[21] and is defined as:

$$nRSS = \frac{\sqrt{RSS}}{\bar{y}} \tag{6}$$

with $\bar{y}$ being the mean value of all ys. From the *nRSS*, we define the average normalized RSS (*anRSS*) over the number of points:

$$anRSS = \frac{nRSS}{l} \tag{7}$$

The *anRSS* represents the mean value of the error per measurement point. The *anRSS* also normalizes the *RSS* to the scale of the measurement values and hence is a good indicator for the quality of the model.

However, `SimAnMo` also integrates two well-established metrics. A metric frequently employed to determine the quality of the fit is R^2. We use the following definition:

$$R^2 = 1 - \frac{\sum_{m=1}^{l} (y_m - f_m)^2}{\sum_{m=1}^{l} (y_m - \bar{y})^2} \tag{8}$$

R^2 normally is between 1, indicating no difference between model and the points, and 0. In no case in this article, R^2 lies outside this range.

Another common metric is the root-mean-square error (*RMSE*). It is often employed to rate the predictive power of a generated model. The *RMSE* is defined as:

$$RMSE = \sqrt{\frac{\sum_{m=1}^{l} (y_m - f_m)^2}{l}} \tag{9}$$

It is on the same scale as the data to be analyzed. However, is relatively sensitive to outliers.[22]

One important point about all metrics mentioned above is that points with higher values have a higher influence on their overall value. Consequently, for the generation of the model, points with higher p-values have a higher influence on the model's shape.

To remove this property, there exists the mean absolute percentage error (*MAPE*) metric:

$$MAPE = \frac{\sum_{m=1}^{l} \frac{|y_m - f_m|}{|y_m|}}{l} \tag{10}$$

It is a relative measurement for the overall error where the individual errors relative to each point are combined equally. In that way, large relative errors at points p with low y-values have the same influence on the overall metric as large relative errors on a p with high y-value. Because of their properties, we expect, in general, the *MAPE* to be higher than the *anRSS* for the same model. The ideal value for both metrics is 0.

In general, the training data has to be carefully collected. This means that no side-effects for a small $p$dominate the actual runtime of the algorithm in that range like initialization procedures or reading data from the file system.

3.3 | Integrated model types

SimAnMo contains three type of model functions. Polynomial logarithmic (pol-log) models (cf. Section 3.3.1), which also cover the special cases of constant runtime, exponential polynomial models (exp-pol) models to cope with exponential growth of the runtime (cf. Section 3.3.2) and factorial runtime models (cf. Section 3.3.3).

3.3.1 | Polynomial logarithmic models

Like Extra-P,[20] SimAnMo can model polynomial runtime behavior. Following the results of Reisert et al.,[23] the shape of those models is given by Equation (11). Hence, the values of c_0, c_1, i, and j are determined by iterative refinement and minimizing the costs of the constant model after i and j are set.

$$c_0 + c_1 \cdot p^i \cdot \log_2^j(p) \tag{11}$$

Since this case was discussed by Burger et al.[19] we omit the details.

3.3.2 | Exponential polynomial models

We propose a new shape for modeling functions for exponential runtime behavior in Equation (12)

$$f(p) = c_0 + c_1 \cdot 2^{k \cdot p^i} \tag{12}$$

The two main ideas behind it are: (1) The enumeration algorithm we already considered demonstrates that the variable in the exponent may grow faster than linearly (quadratic in that case). (2) Previous work about the LLL lattice reduction algorithm[19] shows that the practical behavior of programs with polynomial runtime can be accurately modeled by polynomials with a real exponent,although the complexity class $\mathcal{O}$ for the underlying algorithm is given by integer exponents. For example, the complexity class of an algorithm *ALG1* may be given by $\mathcal{O}(p^3)$, but the model describing its behavior the best is $\alpha_1 \cdot p^{2.61}$ and not $\alpha_2 \cdot p^3$ or $\alpha_3 \cdot p^2$ for $\alpha_1, \alpha_2, \alpha_3 \in \mathbb{R}$.

We again consider Figures 1 and 2. The green curve shows the exp models as used in the extended Extra-P version based on Equation (4). This type of model is also created by SimAnMo since we extended SimAnMo with exp models. To that end, we followed the methods described in Section 4.3. In that way, SimAnMo automatically determines the coefficients. Figure 1 demonstrates that the exp model performs better than the lin-log model. Its graph coincides with our novel model and hits all the training points. However, the value of the prediction for $n = 250$ is too low by 150 orders of magnitude, as observable in Figure 2.

The dashed lines show a lin-log and an exp model, respectively, which employ $n \in \{20, \ldots, 100\}$ as training data. Although those training points also would cause infeasible runtime and the resulting models perform better, they do not give reasonable predictions for $n = 250$. The values are too low by 103 and 70 orders of magnitude, respectively. In contrast, our exp-pol model, employing the small training data range, correctly predicts the function's behavior.

3.3.3 | Factorial models

Compared to the SimAnMo version presented by Burger et al.,[24] we also integrated a new type of model: It is capable of representing factorial runtime behavior. This means that the growth of the runtime is mainly directed by the term $p!$ An example for factorial runtime behavior is finding permutations of a data set or a list. Another example is the traveling salesman problem (TSP). It consist of searching for a tour in a graph which visits all nodes exactly once and the overall length of this tour is minimized. This problem, for example, occurs when routing vehicles or in chip design. The shape for those factorial models is given in Equation (13):

$$f(p) = c_0 + c_1 \cdot p! \cdot p^i \cdot log^i(p) \tag{13}$$

The factorial function $p!$ mandates p to only have integer values. The terms $p^i \cdot log^i(p)$ allow more flexibility for fitting the training data. Their insertion is motivated by our previous results from modeling exponential runtime behavior.[24] Test cases based on these models which justify this

decision are evaluated in Section 5.4. Concerning the rapid growth of the factorial term and taking the results of Figure 2 into account suggests that a lin-log approach fails even more severely in the case of factorial behavior.

4 | THE SimAnMo FRAMEWORK

SimAnMo is written in C++-11 and open software.* It contains features to generate model functions, to visualize the fit to the training data, and to assess the quality of the prediction if additional measurement points are available. The overall process to create the models is semi-automatic. The training data can be generated via scripts. Afterward, the user has to decide which training points are the input for the model generator. Also the model type (pol-log, exp-pol, fac) has to be selected. The rest is automatically performed by SimAnMo without further user intervention. The final result is a pdf file with the diagrams in the same manner as they are shown in this article. The report also contains the values of the quality metrics employed. The printed digits are limited to a reasonable number. Internally all calculations are performed with at least double precision. Additionally, the development of the quality of the model during the iterative process can be visualized.

4.1 | Software architecture

SimAnMo can be employed as a stand alone console application or as a static library alternatively. The behavior of the application is configured via more than thirty command line parameters which are described in the documentation. This includes the configuration for the models themselves like their type or the possible range of their coefficients. Furthermore, users can choose for the configuration of the simulated annealing process the starting temperature, the number of steps per temperature value or the number of threads employed. The only mandatory parameter is, however, the path to the input file with the training data and the optional measurement points. Remaining parameters are experimentally set to default values which result in reasonable behavior in all tested cases. The mandatory input file has to be structured as shown in Listing 1.

```
( p-value of first training data point ; its runtime )
( p-value of second training data point ; its runtime )
...
#ADDPOINTS
( p-value of first measurement point ; its runtime )
( p-value of second measurement point ; its runtime )
...
```

Listing 1: Layout of an SimAnMo input file

This structure indicates that SimAnMo can create other models than those relating the size of one input parameter to the resulting runtime of a program. In principle, every domain in which the relation of the value p to the modeled value y can be expressed by one of the available types of models $f(p)$. An additional model $f(p)$ for the respective domain can also be added.

In its default configuration, SimAnMo searches for a polynomial-logarithmic model that minimizes the *RSS* metric of the training points in the pre-defined coefficient range. If the quality of the model is not sufficient, the best fitting exponential-polynomial model is searched and returned if it is better.

The library version of SimAnMo is mainly based on the function SimAnMo::findModel. It returns a SimAnMo::FunctionModel object. This object contains all the necessary information, like the resulting model, its type and its cost.

SimAnMo::findModel has three parameters as shown in Listing 2. The first two of type std::map<double, double>& give the pairs of p-values and training data points and measurement points, respectively. The third of type std::string corresponds to the input parameters of the console application.

```
SimAnMo::FunctionModel SimAnMo::findModel(
        std::map<double, double>& training_points,
        std::map<double, double>& measurement_points,
        std::string options)
```

Listing 2: Usage of the SimAnMo library

4.2 | Calculating the models

We extended the previous version of the code presented by Burger et al.[19] with the capability to search for exponential models. They possess the shape of Equation (12). We also added factorial models following Equation (13). Due to SimAnMo's software design, the main code structure remains the same when adding new types of models. The quality of the fit of the model to the training data can be assessed by two metrics. By the standard *RSS* and by *MAPE* which was also newly integrated into the framework.

For example., to *RSS* as it was used for the programs with polynomial runtime behavior.[19] In fact, the *RSS* is the standard metric to minimize within the Extra-P application and the traditional lin-log approach. For the other models in this article, we stick to the *MAPE* in Sections 5.2 and 5.4.

SimAnMo's model generation approach is based on parallelized simulated annealing. As discussed in Section 3.1, models are generated in a two step approach. First by iterative refinement and second by setting the values of c_0 and c_1 to minimize the cost metric between the training data points and $c_0 + c_1 \cdot \text{constModel}(p_m)$, $1 \leq m \leq l$, that is, the respective model values. Both steps are implicitly performed in each simulated annealing step.

4.2.1 | Iterative refinement

Iterative refinement is performed within simulating annealing where randomized neighboring models from the current model are generated. To that end, the coefficients within the constant model part, that is, in *i*, *j*, and *k* depending on the model type, are modified by a small randomized adjustment. Boundary checks limit the acceptable values of *i*, *j*, and *k*. The maximum allowed range of the changes was determined experimentally.

4.2.2 | Determining c_0 and c_1

SimAnMo provides two different approaches for setting c_0 and c_1, depending on the cost metric employed.

For *RSS*, after determining the values of the other coefficients in the model by randomization, c_0 and c_1 are calculated by a Jacobi SVD decomposition. The calculation is performed by the Eigen-library.[25] It returns a least squares solution of the overdetermined equation system $Ax = b$ where the vector b contains the y-values of the training data points, that is, the actual runtimes. The matrix A is of dimension $2 \times l$ with l being the number of training points and the first column of A consists of ones. The second column contains the evaluations of the constant models (cf. Section 3.1) at the corresponding p-values. The vector x of length 2 contains the solution, that is, c_0 and c_1 which fulfill the overdetermined system of equations.

The second to determine c_0 and c_1 for *MAPE* is more compute intensive. The main idea is to minimize the *MAPE*-equation with the choices of c_0 and c_1. This is done with the BLEIC (**b**oundary, **l**inear **e**quality-**i**nequality **c**onstraints) algorithm of the ALGLIB-library.[26] The underlying optimization algorithm is a nonlinear conjugate gradient method. It also requires the gradient of the function to minimize which is calculated following the analytical differentiation of this function. Due to its complexity, this second approach may be more unstable than the *RSS*-based variant. For the conjugate gradient method, an adequate stopping criterion is required. At the moment, this is performed by setting the EpsX parameter in the call of BLEIC's minbleicsetcond function to 10^{-9}. That means, the conjugate gradient algorithm stops, when the Euclidean norm $||\mathbf{v}||$ of the scaled step vector $\mathbf{v}$ is smaller than EpsX= 10^{-9}.

In average, the computation with *MAPE* requires ten times as long as the *RSS*-based variant. For example, the models generated in Section 5.4 require between 4 and 7 s for *RSS*-based and 60–75 s for *MAPE*-based. However, tuning the stopping criterion for the conjugate gradient can reduce this difference. Another possibility to improve the performance and to increase the stability is the function minbleicsetscale which sets the scaling of the variables. The scaling is set to [1, 1] by default. This is not optimal since, in particular, in the case of exponential and factorial runtime behavior the difference in the size of the variables is several orders of magnitude.

Switching between the *RSS* and the novel *MAPE* minimization is realized by just adding one command line parameter.

4.2.3 | Optimizations for simulated annealing

We also added two heuristics to the annealing process to improve the convergence and the quality of our models: (1) Backtracking: Each thread protocols whether the modified solution which is generated in the current iteration is better than the former solution. If q_1 (parameter can be set by the user) successive solutions, starting from the last solution S_{imp} which resulted in an improvement in iteration t, are worse than S_{imp}, the actual solution S_{act} in iteration $t + q_1$ is reverted to S_{imp}. The overall model quality reacts in a very sensitive fashion to changes of the values in the exponent. There are apparently many blind ends for the annealing process from which it can only hardly escape. This is simplified by backtracking.

(2) Adaptive random modifications: Depending on the overall allowed search space for each coefficient and its current value, SimAnMo adapts the extent of the modifications to this coefficient when generating random neighboring solutions. Additionally, for exp-pol models (Equation 12), the values for k and i are both modified at once since systematic experiments showed that this method improves the convergence behavior.

4.3 | Integrating new model types

Integrating a new model type into SimAnMo is simplified through a uniform interface. Two basic constructors for the model object are required. The first one is the default constructor returning an empty solution. The second constructor takes given values for c_0, c_1, i, $j$and optionally k and creates a model for those. A third additional constructor generates a random solution based on the input data. Its costs must already lie in a feasible range. Those models are the starting point for the simulated annealing procedure.

Additionally, a routine is required that slightly, randomly modifies a given model for the generation of a random neighbor within simulated annealing. This routine must also check that the returned model is still feasible. Furthermore, two routines to evaluate the model and the constant model at a position p must be provided. For the optional output of the results, SimAnMo requires methods to represent the model as string and as LATE X-formula.

For the case of factorial models, these are less than 400 lines of code, and less than 250 when excluding printing functionality. The main challenge for all integrated model types is the generation of a random feasible solution and the adjustment of the appropriate extent of the random modifications.

4.4 | Numerical aspects

Most internal calculations concerning the quality metrics are performed with double precision. The only exception is the calculation of $p!$ which is done with 64 bit integers. To avoid numerical problems, following measures and experiments are taken:

1. We limit the range for the coefficients c_0 and c_1 in Equations (11)–(13). This avoids that they become arbitrary small or arbitrary large and that the models overfit. The default limit is 10^{-11} and 10^{40}, respectively, which worked for all cases we considered. Hence, we are confident that those experimentally defined bounds will also work well for other models. If a value is smaller or higher than the limits, it is set to the limit. Then, the costs are re-calculated before returning the model. Additionally, a warning code is returned by the routine which determines c_0 and c_1 to indicate the overflow.
2. When c_0 and c_1 should be determined with the *MAPE*-approach, first the coefficients are set with the *RSS*-approach. In the case that a warning is returned (as explained above) the *MAPE*-approach is stopped. This is done because the chance is very high that the conjugate gradient will not converge or will return a very bad result. An error is returned that signals the caller that for this choice of i, j, and k no adequate c-values can be found and that i, j, and k have to be changed.
3. When modifying or randomly creating a solution, the annealing process checks whether its costs are feasible before taking the solution into account. Feasible means in that context that the *RSS*or the *MAPE* are not allowed to be many orders of magnitude higher than the (squared) sum of all measurement values. This prevents randomly accepting a very bad solution from which it is hard to come back to the range of good solutions in small steps.
4. For the *MAPE* values, we compared the convergence behavior of simulated annealing when rounding *MAPE* to eight significant digits and without rounding. No instabilities can be observed in the second case. Hence, by default, rounding is not performed. For *RSS* we do not observe numerical issues even for very large values or very low values.

5 | EVALUATION

In this section, we evaluate the quality of SimAnMo's generated models for the three different types of models presented in Section 3.3. We start with the polynomial logarithmic models. Afterward, we discuss the exponential polynomial models beginning with a test suite of synthetic tests. Those provide the advantage that we can control the parameters and have knowledge about their real behavior a priori. Additionally, four real world programs with exponential runtime behavior are analyzed. Finally, we discuss the factorial models with one theoretical example and two real world problems. For all models, we show that our approach works well for ARM architectures, too. We assess the quality of the fit to the training data, in general, at hand of the *RSS* and the *MAPE* metric. In some cases, we additionally employ the *anRSS* to highlight some details. As mentioned in

Section 3.2, SimAnMo also calculates *RMSE* and R^2. Since they offer no additional insights in the cases investigated, we do not list them for clarity of the results and focus on the others.

5.1 | Methodology

We perform tests on three different ×86/×64 hardware platforms. *SystemAMD* has an AMD Ryzen 3900X (12 cores, 24 threads, 64 MB of L3 cache) and 32 GB RAM. It operates Windows 10 (version 20H2) and employs the compiler of Microsoft Visual Studio 2019 (version 16.8.4).

SystemHaswell has two Intel Xeon E5-2680 v3 processors with 12 cores and deactivated Hyper-threading. Up to 768 GB RAM are available. The OS is CentOS 7.2 and the compiler employed is g++ (version 8.3).

SystemCascade has two Intel Platinum 9242 Processors (48 cores, 48 threads, 71.5 MB of Smart Cache each) and 384 GB RAM. The operating system is CentOS 8 and the compiler employed is g++ (version 10.2).

The Intel systems are nodes of the Lichtenberg Supercomputer at Technical University of Darmstadt and the scheduler allows jobs with a runtime up to 24 h.

We also employed Raspberry Pi 3B (4 cores, 4 threads, 1 GB LPDDR2 SDRAM) and Raspberry Pi 4B (4 cores, 4 threads, 8 GB LPDDR4-3200 SDRAM) single-board computers to represent the ARM architecture. Both are running CentOS 7 (AltArch) and the codes have been cross-compiled with g++ (version 9.1). Hence, we demonstrate SimAnMo's applicability on a broad range of systems.

5.2 | Polynomial logarithmic

As demonstrated by Burger et al.,[19] SimAnMo creates reliable pol-log models for different implementations of the LLL lattice reduction algorithm with various types of input lattices. The models generated are much more accurate than the existing theoretical models. To underpin the previous results,[19] we evaluate the dense matrix matrix multiplication as one additional application with pol-log runtime behavior.

Because of the number of primitive arithmetic operations the runtime is expected to grow in $\mathcal{O}(p^3)$. We implemented a naive multiplication without optimization techniques like blocking. As data type, we employ cpp_dec_float_50 from boost::multiprecision. It guarantees a 50 decimal digit precision (double precision guarantees 15 digits). This choice increases the demand for memory and includes a non-standard data type into the runtime model. We added an OpenMP parallelization for the outermost for-loop. It runs over the rows of the left matrix and the columns of the right matrix. Thus, the model includes the effect of employing multiple threads. We used 12 threads in this case on *SystemAMD*. The results are summarized in Figure 3.

The model fits the training data $p \in \{550{,}600, \ldots, 1050\}$ very well. The prediction for all measurement points is accurate with a deviation of only 2.1% at $p = 5750$ (1788 s vs. 1827 s). To asses the quality of the prediction in an additional way, we perform a so called *robustness test* where we create a model of the same type by employing all available further measurements for the model generation. Comparing the deviation in the coefficients of this robust model to the original model is an indicator for the quality. A high stability of the corresponding coefficients indicates a good model.

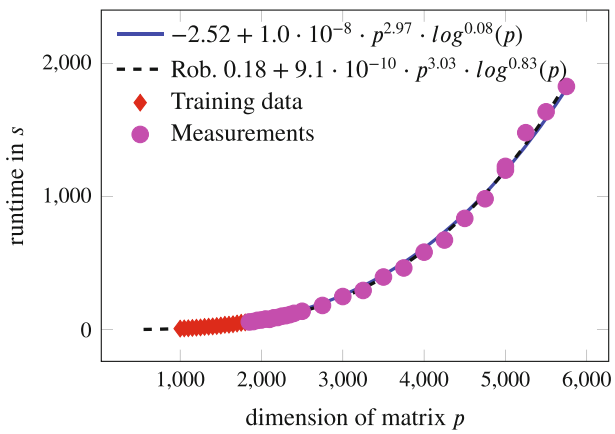


FIGURE 3 Models for parallel matrix matrix multiplication on ×86. *RSS*=0.18 and *MAPE* = 0.022

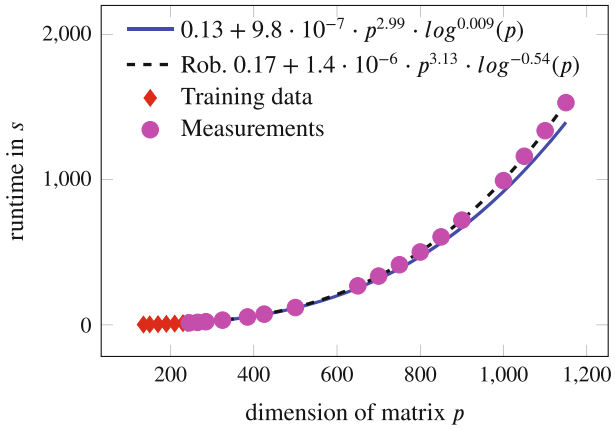


FIGURE 4 Models for parallel matrix matrix multiplication on ARM. *RSS*=0.11 and *MAPE* = 0.007

The robust model for the parallel matrix multiplication nearly coincides with the original model. The exponent of the dominant polynomial factor only varies by 0.06 which is very stable. The actual growth of the runtime is in $\mathcal{O}(p^{2.97})$ and $\mathcal{O}(p^{3.03})$, respectively. This is identical to the prediction from theory, although a special data type with additional management overhead is employed. We want to highlight that for the matrix matrix multiplication the choice of the training data was manually refined. Because for smaller values the resulting model grows too slowly. Cache effects distort the real behavior in this region although cpp_dec_float_50 is four times more accurate than double precision. However, the cache (64 MB in the case of *SystemAMD*) is large enough to keep large parts of the matrices for smaller p-values.

We also performed the matrix matrix multiplication on the Raspberry Pi 3B. Figure 4 shows the model generated by `SimAnMo` in relation to the data points. The model again fits the training data $p \in \{135,140, \ldots, 240\}$ well. It predicts the highest value tested, $p = 1150$, with an error of 8.8%. Because of the much smaller cache size on the Pi 3B of just 512 kB, there are much fewer side effects for smaller problem sizes. Hence, already from $p = 135$, the training points are the base for a reliable model. Since the error at the last measurement point is slightly bigger on the Pi 3B the robust model fits the higher points better. However, the difference of the exponent in the polynomial factor of 0.14 is still small.

In both cases of multiplication discussed, `SimAnMo` reliably predicts values for p which are about five times higher than the last training data point.

This example and those in earlier evaluations[19] highlight that `SimAnMo` generates reliable pol-log models with comparable or better accuracy than existing solutions.

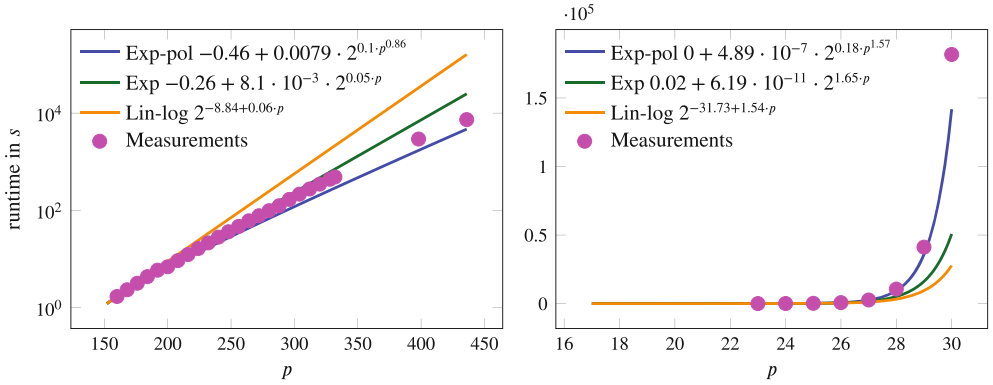
5.3 | Exponential polynomial

Here, we demonstrate the superiority of `SimAnMo`'s exp-pol models compared to the existing approaches linear-logarithmic regression and exponential models as employed in Extra-P.

5.3.1 | Synthetic tests

We employed a C++ code which performs a fixed series of calculations on randomly chosen elements of a 50,000 × 50,000 matrix filled with random double values. The number of the series performed is dependent on a formula $f(p)$ that grows exponentially in p. The different $f(p)$-functions are given for the corresponding test case in the first data row of the table in Figure 5. For each value of p, five iterations are performed and a small random jitter to the result of $f(p)$ between $\pm 2.5\%$ is added. We omit the visualization of the fit of the model to the training data since the different lines for the models nearly coincide.

The $f(p)$ of test *Synthetic 1* is motivated by the Grover quantum search algorithm.[27] This algorithm allows to reduce the complexity of searching an entry in an unsorted map with N elements from $\mathcal{O}(N)$ on a classical architecture to $\mathcal{O}(\sqrt{N})$ on a quantum computer. We imagine an algorithm whose number of repetitions grows exponentially in the complexity of searching in an unsorted table with p entries. That means $2^{\mathcal{O}(p)}$ on traditional hardware and $2^{\mathcal{O}(\sqrt{p})}$ on a quantum computer, respectively. The relative differences in *MAPE* are higher than those for *RSS*and the exp-pol model provides the smallest value. The prediction of the exp-pol model is also the best which is shown on the left side of Figure 5 on a logarithmic scale. The exp-pol model slightly underestimates the real value, while the lin-log model overestimates it by more than one order



	Synthetic 1	Synthetic 2
$f(p)$	$2^{1.49 \cdot \sqrt{p}+2.25}$	$2^{0.11 \cdot p^{1.71}+1.12}$
Training	$p \in \{148, \ldots, 157\}$	$p \in \{17, \ldots, 22\}$

	Exp-pol	Lin-log	Exp	Exp-pol	Lin-log	Exp
RSS	0.000628	0.000639	0.000620	0.000210	0.077150	0.000665
MAPE	0.005499	0.005707	0.005580	0.030882	0.037379	0.083713

FIGURE 5 Two synthetic examples of exponential runtime behavior

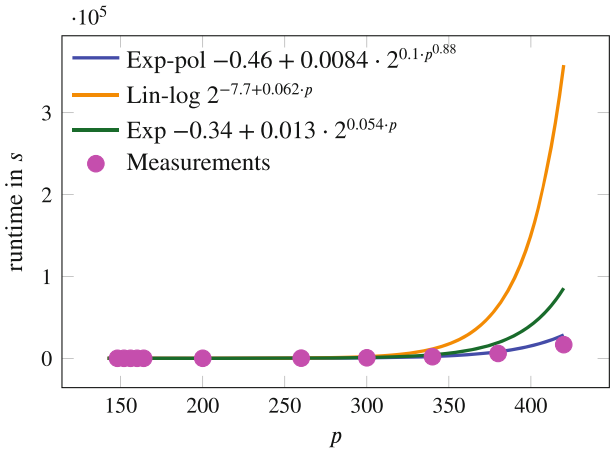


FIGURE 6 *Synthetic 1* model on Raspberry Pi 4B

of magnitude. The exp model lies between lin-log and exp-pol. In that case, the lin-log model lacks the flexibility to cope with the exponent 1/2 of p.

Synthetic 2 tests an exponent of p which is greater than 1 in practice. This can mainly be noticed by the drastically increasing runtimes when increasing p. While the first training point at $p = 17$ only requires 0.025 s of runtime, the highest measurement at $p = 30$ already requires 181,738 s, that is, more than two days. The *RSS* value for lin-log is about 35 times higher than for exp-pol since the curve can not fit the last training point $p = 22$. The evaluation of the models on the right side of Figure 5 shows that the prediction of the exp-pol model is the most accurate. It underestimates the measurement by less than 20%. At the same time, the prediction of the lin-log model and the exp model is too low by a factor of 7 and 3.5, respectively.

Finally, we also employed the Raspberry Pi 4B. We repeated the test *Synthetic 1*.. The model in Figure 6 is generated from training points $p \in \{138,139, \ldots, 147\}$ since the runtimes are much higher than on the desktop system.

However, the resulting situation is very similar to the ×86 system. The lin-log model considerably overestimates the measurements, while the predicted values of the exp-pol model slightly overestimate them. For $p = 420$, the prediction is too high by 20%. In contrast, the lin-log model's value is too high by a factor of 20 and the exp model's by a factor higher than 5. Hence, this example again demonstrates that our model generation approach is directly applicable to different architectures.

We summarize that we are confident that the additional degree of freedom in the exponent resulting from Equation (12) considerably increases the quality of the models generated compared to the exp model. Hence, we do not further consider the exp model in the following. In particular, it is just a special case of the more generic exp-pol model. The lin-log models, in general, have problems to predict the runtimes of functions where p has an exponent $\neq 1$.

5.3.2 | SVP and methodology

Three programs we employ for our further evaluation are so-called SVP solvers. They try to find the shortest non-zero vector in a lattice.[6] The basis $\mathbf{B}$ determining a lattice consists of d basis vectors of length n. n is also called the dimension of the lattice and d its rank, respectively. In our test cases n always equals d. The SVP is known to be NP-hard for randomized reductions[28] and is expected to be not efficiently solvable by quantum computers[29(p. 151)]. The hardness of the problem and the runtime to solve SVP grows exponentially with the dimension of the lattice. The growth of the complexity when increasing the dimension is mainly estimated theoretically for the different SVP solving algorithms.[30] The input parameter to generate the runtime models is the dimension of the lattice n.

Practically determining a runtime model is complicated by the exponential runtime behavior. Training data points can only systematically be generated for low dimensions and verifying measurements for high dimensions are infeasible. Additionally, the RAM requirements for the sieving algorithms are very high (up to 1 TB). This excludes our ARM system for the tests.

A second challenge is that all SVP solvers are heuristic algorithms. They contain probabilistic elements, for example, sampling random vectors or applying small transformations to lattice bases with random unimodular matrices. Hence, the runtime of each algorithm to solve a problem instance varies for each attempt. This variation can span more than one order of magnitude.[5,31] Thus, at least 20 runs are performed for the same configuration determined by the tuple (algorithm, n, random seed of lattice) and averaged.

The third challenge is that the practical hardness of the problems varies also with the different instantiations possible for each dimension. This includes the type of the random lattice but also different random seeds for the same type.[31,32]

The input bases we employ come from the Darmstadt SVP challenge.[†] We employ the lattice generator provided at the challenge homepage.[‡]

5.3.3 | Results for SVP solvers

In the following, we compare the exp-pol models created by `SimAnMo` with the lin-log model approach on the SVP solver codes. Additionally, we again perform robustness tests.

5.3.4 | SubSieve

SubSieve[4] considerably reduces its runtime and memory-consumption compared to previous sieving variants. This is because it only requires to solve $n - \alpha$-dimensional ($\alpha \in \mathbb{N}$) sub-SVPs to solve the overall n-dimensional SVP.

To generate the training data, we performed 100 runs for each $n \in \{70, 72, \dots, 80\}$. The 100 runs were evenly distributed to random lattices with five different random seeds. For each dimension, the average of the runtimes is the training point. We passed the length of the shortest vector to SubSieve to indicate termination. SubSieve is a serial code. Our resulting model is compared to the lin-log model in Figure 7. Both curves nearly coincide and fit the training data well with the lin-log model slightly missing $n = 82$. The quality metrics are better for the exp-pol approach with $RSS = 98.1$ / $anRSS = 0.016$ / $MAPE = 0.051$ and $RSS = 258.3$ / $anRSS = 0.026$ / $MAPE = 0.055$ for the lin-log model, respectively.

For the evaluation, we continued the scheme of 100 runs per dimension with five different seeds for each dimension up to $n = 90$. Additionally, for $n = 94, 96, 98$ ten runs with seed 0 were performed. For higher dimensions, the runtime is expected to exceed 24 hours in most cases which hinders a systematic evaluation.

The resulting prediction is shown in Figure 7. In general, the exp-pol model has a higher accuracy when predicting the measurements than the lin-log approach. However, there is an exception at $n = 96$. In that case, the problem seems to be rather easy since the time to solution is not much higher than for $n = 94$. Furthermore, the data point lies outside all possible curves of exponential functions that try to fit all data points. The robustness test shows stable coefficients for the model.

5.3.5 | G6K

The General Sieve Kernel (G6K)[5] is the fastest known SVP solver. It holds the first places in the Darmstadt SVP Challenge.

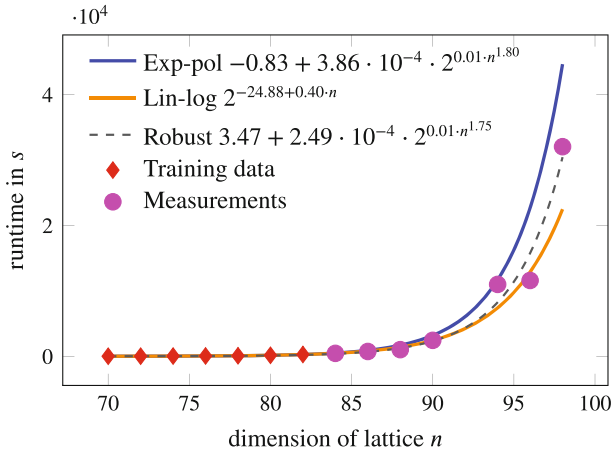


FIGURE 7 Prediction of both SubSieve models

To create a model for G6K, we first generated the training points on *SystemHaswell* with 24 threads for $n \in \{90,100, 110,120, 130\}$. The G6K code is configured to solve the SVP challenge, that is, to find a vector shorter than the given bound. We choose five different random seeds for each dimension and executed three runs per seed.

In Figure 9, the exp-pol graph again fits the training data very well. The lin-log model misses the last point. This visual impression is confirmed by the quality metrics. While the exp-pol model results in $RSS = 520.1$ / $anRSS = 0.0044$ / $MAPE = 0.0081$, the lin-log model has considerably higher values of $RSS = 1{,}806, 679$ / $anRSS = 0.263$ / $MAPE = 0.275$. Hence, the exponent which grows linearly in n is not able to fit the distribution of the training data. In contrast, the exp-pol provides the flexibility required.

For further measurement points, we rely on the SVP challenge entries related to Albrecht et al.[5] and the number of threads given in the hall of fame. We performed a scaling analysis for the code on our system and extrapolated the scaling behavior to the number of cores and threads given in the SVP challenge entries. Implicitly, we use the fact that our system runs with about the same clock speed and that the processors employed have no big differences in their architecture. So the only variable considered is the number of threads employed for the respective run. In that way, we get measurement points for up to dimension 155. There, the time to solution was more than 14 days and a total CPU time of 1056 days was spent. This procedure may introduce some errors and the entries in the hall of fame result from only a single run.

Nevertheless, our comparison of models and their predictions in Figures 8 and 9 show that the training data and the extrapolated measurement points follow a sensible pattern. We see that both models underestimate the last two points $n = 153$ and $n = 155$. The prediction is too low by a factor of 1.33 for $n = 153$ and 1.14 for $n = 155$, respectively, for the exp-pol model. The lin-log model is much further away from the real values and underestimates the runtime by a factor of 11.63 for $n = 153$ and 11.90 for $n = 155$, respectively. In that case, the difference in the prediction for the last measurement point $n = 155$ for both models is already about one order of magnitude. It quickly grows to several orders of magnitude when further increasing n. Again, the robustness test shows stable coefficients.

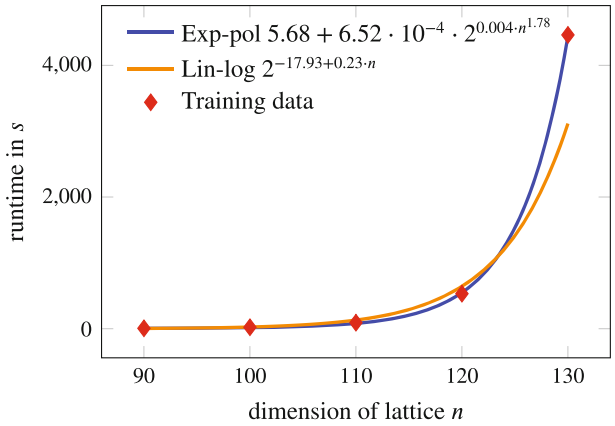


FIGURE 8 Fitting of the G6K models for the training points for $n \in \{90,100, 110,120, 130\}$

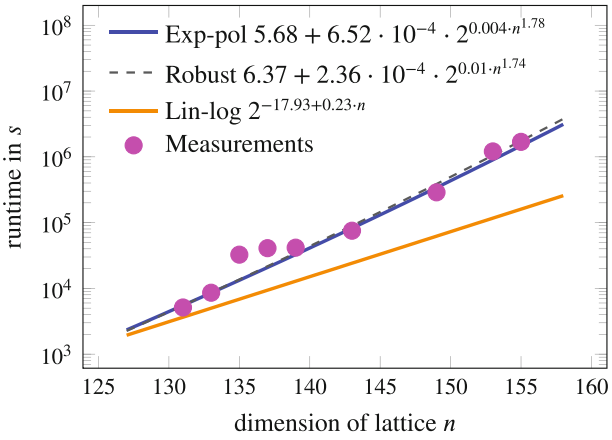


FIGURE 9 Prediction of both G6K models

5.3.6 | Enumeration with extreme pruning

Our last test case is an SVP solver based on enumeration with extreme pruning. Those algorithms build a search tree for all possible coefficients for the shortest vector. They heuristically cut off large parts of that tree where the chance is low to find the shortest vector.[9,31] If the shortest vector was removed, the procedure has to be repeated with a slightly shuffled lattice basis.[6] Hence, several heuristics and random elements are included in the algorithm. They result in highly varying runtimes for the same problem when solving it multiple times.[31] The implementation employed for our experiments is the progressive BKZ (pBKZ) library. It presented by Aono et al..[6]

We employ the first part of the runtime data presented by Aono et al.[6] to predict the remaining points. The runtimes for the training data $n \in \{102,104,106,108,111\}$ and the models created are summarized in Figure 10. The data has an outlier for $n = 106$. From the diagram it is not obvious to decide which graph fits the training data better. It is even ambiguous when considering the quality metrics. The *RSS* for the exp-pol model is $2.83 \cdot 10^8$, while it is $1.35 \cdot 10^8$ for the lin-log model. However, the *MAPE* of the exp-pol model is 0.48 (*anRSS* = 0.26) and better than the *MAPE* of the lin-log model which is 0.59 (*anRSS* = 0.18). This behavior could only be observed for this test case. The distribution of the training data reflects the highly random nature of the algorithm. Aono et al. only performed single runs for their data points.

Figure 11 shows our evaluation. We also include the three points for $n \in \{138,142,146\}$ from Reference 6. They are taken from the Darmstadt SVP challenge hall of fame. We employ them to validate our model against higher dimensions. The lin-log model lies further away from the measurements than the exp-pol model in all cases but for $n = 112$ and $n = 123$. For $n = 123$, the exp-pol model predicts a considerably higher runtime compared to the measurement and the lin-log model. The data suggest that $n = 123$ was one of the runs which only require few randomization of the input base. It is much faster than the run for $n = 119$ and about equally as fast as the run for five dimensions less at $n = 118$. The fact, that the lin-log model underestimates all measurements after $n = 123$ also is an indicator for this assumption.

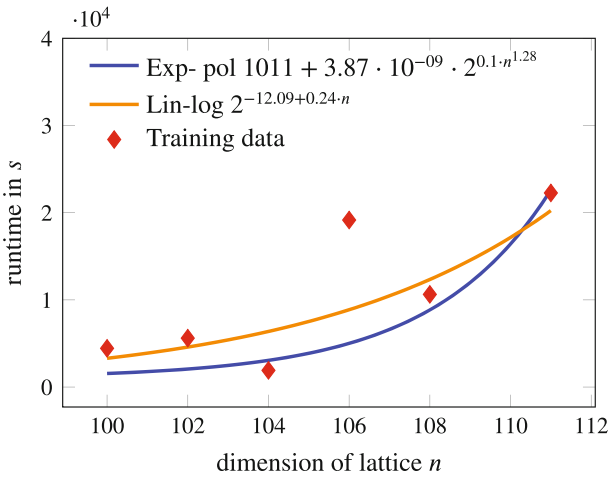


FIGURE 10 Exp-pol model and the lin-log model for training points $n \in \{102,104,106,108,111\}$

In green, we see the model from Aono et al. which was created by the data of their cost-simulator for the runtime of progressive BKZ. Simulation points were created for up to $n = 150$ in steps of five. Their evaluation shows that the simulator gives very good estimates for the runtime of their pBKZ library for SVP challenge problems. The line is just a linear regression to the logarithmized measurement points.

We see that the model which `SimAnMo` creates from $n \in \{102,104,106,108,111\}$ gives a good prediction for the real time to solution up to $n = 146$.

5.3.7 | Factorizing product of prime numbers

Finally, we analyze a different problem from the cryptanalysis context which is the factorization of big integer values. To that end, we generate two prime numbers p and q which fulfill several properties that are recommended for the Rivest–Shamir–Adleman (RSA) cryptosystem.[33] For example, $|\log q - \log p| > 0.1 \ \wedge \ |\log q - \log p| < 0.3$. We calculate the product $n = p \cdot q$ and then perform a OpenMP-parallelized brute force attack which tries to guess the right p. The data type employed is cpp_int from the boost library. The input parameter p is the bit-length that is employed for the independent_bits_engine random number generator within the prime number generator that is based on the Miller–Rabin primality test. The tests are performed on *SystemCascade* and the runtimes for a fixed p have a very high variance. For example, for $p = 42$, the fastest run is 30 times faster than the slowest. Figure 12 shows that the fit to the training data of our model and the lin-log model is quite good. The *RSS* of the lin-log model is even somewhat better (1378 vs. 1628) but its *MAPE* is worse (0.100 vs. 0.083).

The predictive quality is analyzed in Figure 13. Both types of models give good estimates for the real runtime. The error at the last measurement point $p = 49$ is 16% for our model and 21% for the lin-log model. In this case, both models work well.

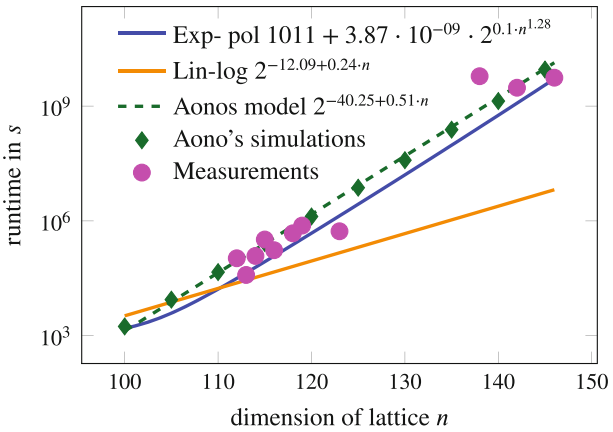


FIGURE 11 Prediction of pBKZ models

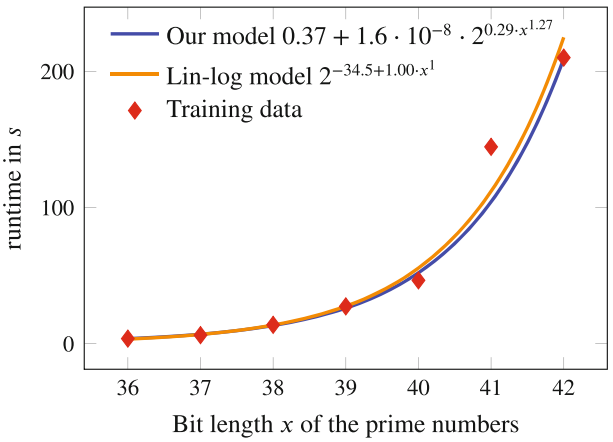


FIGURE 12 Exp-pol model and the lin-log model for training points $x \in \{36, \ldots, 42\}$

We summarize that the exp-pol models deliver the better approach to describe the training data in all of our test cases. For SubSieve, the differences between the exp-pol model and the lin-log model are less significant than for G6K-Sieve and the pBKZ library. The experiments show that the exp-pol approach is more flexible than the lin-log models when trying to accurately model the behavior of the training data. Due to the nature of exponential functions, the number of measurement points for verifying the predictions is limited. However, the experiments performed strongly indicate that the predictions of the exp-pol models are more realistic than previous approaches.

5.4 | Factorial polynomial

We start the evaluation of the new model for factorial runtime behavior with a synthetic test-case, which is $f_{synth}(p) = 20 + 0.13 \cdot p!$ As training points we employ $p \in \{1, 2, \dots, 6\}$ and let the model predict up to $p = 20$. The results are shown in Figure 14 assuming a theoretical runtime in nanoseconds. A stair like pattern on a logarithmic y-axis of the model reflects the factorial behavior. The coefficients i and j quickly converge to 0 and the resulting model predicts $p = 20$ with an error of 0.46%. For $p = 21$, the value of $p!$ exceeds the range of 64 bit integers and gives an impression how fast the factorial runtimes grow even in comparison to (super-)exponential functions discussed above. In this theoretical case we included a lin-log model in orange. As derived from the theory, the lin-log approach is unable to fit such a rapid growth and fails in giving meaningful predictions or even fitting the training data. Hence, we omit it in the following evaluation to keep the diagrams clearer.

Our second test case evaluated in this section is the permutation of an std::string. We internally employ the std::rotate function to modify the order of the strings and perform all permutations recursively. Our test string is *ABC* … *XYZ* truncated to the respective target length

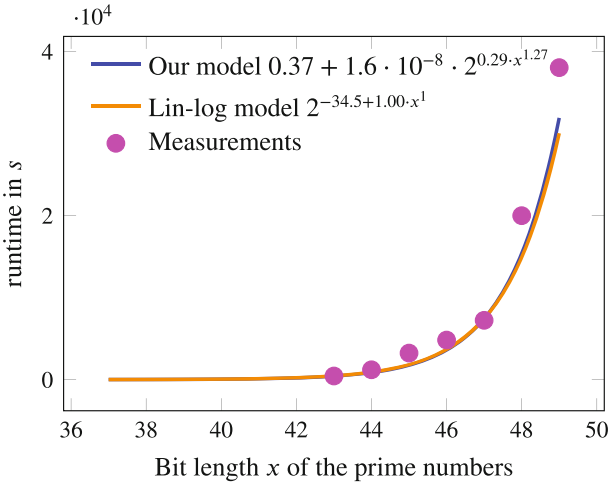


FIGURE 13 Models for factorizing products of large prime numbers

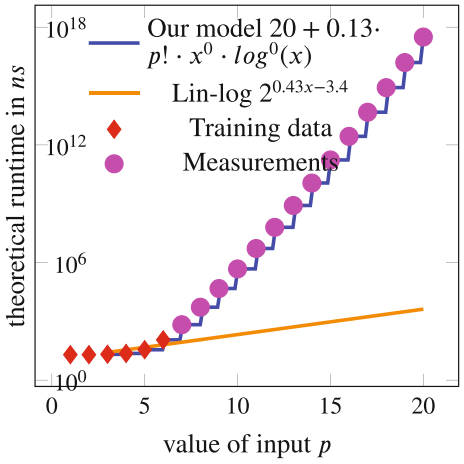


FIGURE 14 Model for $f_{synth}(p) = 20 + 0.13 \cdot p!$ *RSS*=0.24 and *MAPE*=0.002

between 3 and 14. Figure 15 summarizes the results of our experiments for $p \in \{4, 5, \ldots, 10\}$ as training data. Here, the *RSS* of 46,822 seems to be high but we have to consider that the runtime is given in microseconds. For $p = 10$, about 18,500 microseconds are required, which relativizes the *RSS*-value and the small *MAPE* shows that the fitting of the model is really good.

In that case, a small additional polynomial term occurs in the model and the model can predict four additional measurement points correctly, for example, $p = 14$ with an error of 7.9% although the last training point only requires 0.02 s while the last measurement point requires about 1.5 h.

The string permutation test was repeated on the Raspberry Pi 3B and the results are shown in Figure 16. The prediction for the last measurement point is too low by 33% which is very accurate when taking into account that the difference in the runtime between the last training point $p = 10$ and $p = 13$ is more than three orders of magnitude. In this test, the quality of the model differs most when changing the procedure to determine c_0 and c_1 from the *RSS*-based to the *MAPE*-based mode (cf. Section 4.2). Hence, Figure 16 also visualizes the *MAPE*-based graph in orange. In that case, the error at $p = 13$ is reduced to 9%. Furthermore, the fit to the first training points is better for the *MAPE*-based graph. This is mainly due to the high $c_0 = 12.4$ resulting from the *RSS*-based approach. Since the absolute values in the region for $p \in \{5, 6, 7, 8\}$ are considerably smaller than 10 they do not have a large influence on the overall *RSS*-cost metric and the algorithm focuses on fitting the higher training data points. This is a vivid example of how `SimAnMo` can be configured for the desired tradeoff between runtime and model quality. The *MAPE* of the fit to the training data is reduced from 22.9 to 0.18 while the runtime increases from 5.1 to 61 s.

Our last experiment was performed with a C++-code to solve the TSP.[§] It was executed on *SystemCascade*. We focused on the brute force solver which enumerates all tours in a systematic fashion. For the first benchmark, we deactivated the integrated optimization that following a path is aborted if the actual length is longer than an already registered path which visits all nodes once. Figure 17 summarizes the results.

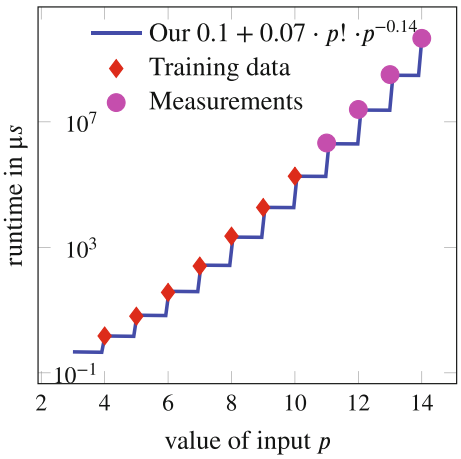


FIGURE 15 Model for string permutation on ×86. *RSS*=46,822 and *MAPE*=0.04

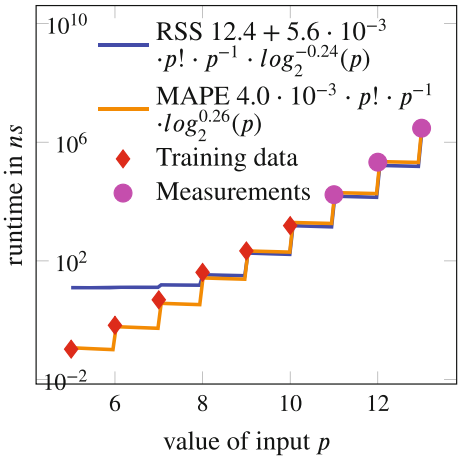


FIGURE 16 Model for string permutation on Pi 3B. *RSS*=1525 and *MAPE*=22.9

SimAnMo finds a well fitting factorial model for $p \in \{8, 9, \ldots, 13\}$ which is able to predict the runtime for $p = 15$ with an error of 7.6%. Higher values for p would last more than one day.

For a second benchmark, we re-activated the early exit and parallelized the brute force solver's execution. The parallelization is realized with OpenMP tasks on the decision for the second node in the path, that is, after we set the first node (the one following the starting node). All possibilities for the next node are executed in parallel. The length of the shortest tour found so far is shared between all threads and write-protected by a critical region. The parallel execution with 24 threads leads to varying runtimes between different executions even if the distance map between the nodes is equal. This lies in the order in which tours are processed and, hence, the order in which the upper bounds for the overall length are updated. Our approach is to re-create a random distance map for each execution and to perform twelve runs for each choice of p. For the training data points and the additional measurements, we employ the average values.

The code for the TSP solver can be found in SimAnMo's repository. Figure 18 summarizes the results. Interestingly, the best model that can be found is an exp-pol model which predicts $p = 21$ with an error of only 3.5%. For $p = 22$, we also started 12 runs. Nine finished within the hard limit of 24 h while three exceeded this boundary and were aborted by the scheduler. Concerning the ranges from the fastest to the slowest run per p value, this distribution for $p = 22$ is expected since the maximum runtime for $p = 22$ can be estimated to 31–33 h. This additional maximum point is drawn in Figure 18 which also shows that the exp-pol model well predicts the average point at $p = 22$.

The best fitting factorial model is shown in dashed. It only approximately hits $p \in \{13, 20, 21\}$ while already overestimating $p = 21$ by 95% (17,808 s vs. 9090 s). The model also hits several boundaries for the ranges of coefficients like -2 for i and $j$as well as 10^{-11} for c_1 which renders the existence of a useful factorial model very unlikely. Even relaxing these boundaries further does not improve the quality of the model.

Hence, the practical behavior of this TSP-variant seems to be described the best by an exponential function and not by an factorial as it was expected.

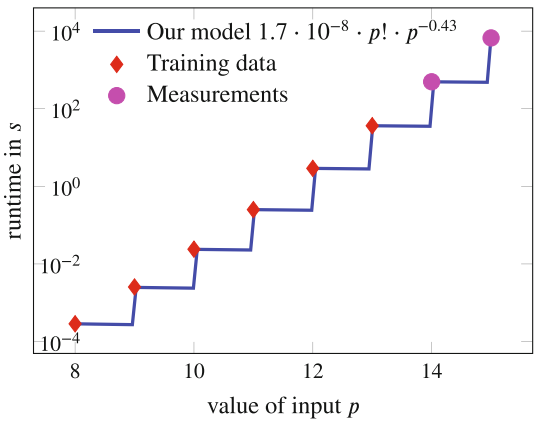


FIGURE 17 Model for the serial TSP. *RSS*=0.000 and *MAPE*=0.005

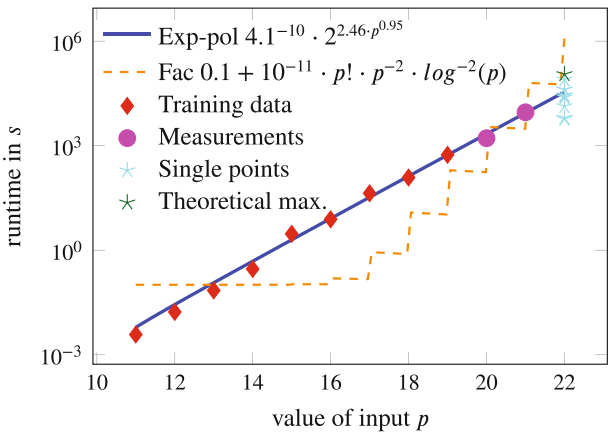


FIGURE 18 Model for early-exit, parallel TSP. *RSS*=57,675 and *MAPE*=13.7

This section demonstrates that the novel factorial type works well for modeling algorithms with factorial runtime behavior. The easy integration of this new models also underpins the flexibility of SimAnMo's software design.

6 | COMPARISON OF QUALITY METRICS

In this section, we analyze the effect of using *MAPE* as metric to minimize by the simulated annealing process. To that end, we additionally generated models for the practical examples from Section 5 which minimize the *RSS* of the model and the training points. We also include another sieving algorithm, called HashSieve, in the considerations since required training points and measurements are already available.[24] The progressive BKZ example is not considered here because of its drawbacks already discussed.

Then, we took the graphs of the resulting prediction and calculated several quality metrics with regard to the measurement points. The results for the exponential runtime examples from Section 5.3 are summarized in Table 1. For each algorithm, the left column shows the metrics resulting from optimizing the *MAPE* to the training data points. In the right column, the respective metrics if SimAnMo minimized the *RSS* are shown. The metrics evaluated in the first three rows are *RSS*, *MAPE*, and *RMSE*. The row labeled with *Deviation* gives the deviation between the biggest measurement and the respective value predicted by the model. The last row shows the highest measurement so that *RSS* and *RMSE* can be set in relation to the overall scale of the values.

For three of the four examples (factorizing, HashSieve, SubSieve), we see that optimizing *MAPE* results in much better models. When considering factorizing or SubSieve we see that the resulting *RSS* is two orders of magnitude lower when optimizing *MAPE*. The *MAPE* and *RMSE* are also lower. Only for G6K Sieve, optimizing for *MAPE* results in slightly worse quality metrics. The deviation at the last point is even better when optimizing *MAPE*, but the prediction for $n \in \{153{,}155, 156\}$ is slightly worse compared to optimizing *RSS*.

The second Table 2 is structured analogously to Table 1. It contains practical examples from Section 5.4with factorial runtime: Creating all permutations of a string and the exact solution to the TSP.

For the string permutation, the minimization of the *MAPE* considerably improves the predictive quality of the models. For example, the resulting *RSS* is reduced by two orders of magnitude and the *RMSE* by a factor of 10. In contrast, for the case of the TSP, the *RSS*-approach results in a slightly better model.

To sum up, minimizing the *MAPE* instead of *RSS* to the training points does not always give the best model. However, in the case that the *MAPE* model is worse, the difference to *RSS* is comparatively small. In the opposite case when the *MAPE* gives the better model, this model can be considerably better than for *RSS*. Hence, if the runtime for model creation is not crucial, minimizing *MAPE* should be the preferred approach.

TABLE 1 Comparison of the quality metrics for the examples of factorizing big numbers, G6K Sieve, HashSieve, and SubSieve

	Factoring		G6K		HashSieve		SubSieve	
	opt. *MAPE*	opt. *RSS*	opt. *MAPE*	opt. RSS	opt. *MAPE*	opt. *RSS*	opt. *MAPE*	opt. *RSS*
RSS	$3.22 \cdot 10^7$	$1.03 \cdot 10^9$	$6.99 \cdot 10^{12}$	$4.91 \cdot 10^{12}$	$6.82 \cdot 10^6$	$1.04 \cdot 10^7$	$3.90 \cdot 10^7$	$1.44 \cdot 10^9$
MAPE	0.19	0.54	0.36	0.35	0.20	0.26	0.22	0.73
RMSE	2145	12,103	706,413	591,353	1502	1866	2360	14,349
Deviation to last	9.9%	74%	5.1%	64%	30%	37%	13%	100%
Last point	38,028	38,028	$1.78 \cdot 10^6$	$1.78 \cdot 10^6$	8141	8141	32,035	32,035

TABLE 2 Comparison of the quality metrics for the examples permutation of a string and the TSP

	String permutation		TSP	
	opt. *MAPE*	opt. *RSS*	opt. *MAPE*	opt. *RSS*
RSS	$2.43 \cdot 10^{14}$	$2.33 \cdot 10^{16}$	$2.33 \cdot 10^6$	$9.73 \cdot 10^5$
MAPE	0.01	0.03	0.17	0.09
RMSE	$7.80 \cdot 10^6$	$7.64 \cdot 10^7$	1080	680
Deviation	0.3%	60%	23%	41%
Last point	$4.59 \cdot 10^9$	$4.59 \cdot 10^9$	6685	6685

But we want to highlight that in all cases our exponential-polynomial and factorial models have a much better predictive quality than the traditional linear-logarithmic approach.

7 | CONCLUSION

We presented the new features of SimAnMo. It is a highly configurable and flexible generator for models of a program's runtime depending on the size of an input parameter p. The SimAnMo-framework is based on parallelized simulated annealing. Its software design makes it easily extendable with new types of models as we highlighted at hand of the factorial runtime models or with new types of cost metrics as we showed for the *MAPE* metric.

The accuracy of the models generated was demonstrated on many theoretic and practical examples on different hardware architectures. The evaluation of the exponential runtime models shows their superiority over the linear-logarithmic approach and over exponential models as employed in an extended version of Extra-P. Our exp-pol models are based on the modeling function $f(p) = c_0 + c_1 \cdot 2^{k \cdot p^i}$ relating the size of p to the actual runtime. Our novel models meet the runtime behavior of the codes in practice. For the polynomial runtime behavior, we extended previous results from cryptanalysis[19] to the application domains of dense linear algebra where SimAnMo also generates well fitting and reliable models. The novel factorial models $f(p) = c_0 + c_1 \cdot p! \cdot p^i \cdot log^j(p)$ are a good example for the easy integrability of new models. They also highlight that reliable models can be generated for this type. With now combining polynomial, exponential, and factorial runtime behavior, SimAnMo covers all important types of runtime complexity.

In the future, SimAnMo will be integrated into the tool PIRA.[34] PIRA automatically refines the instrumentation for profiling in a given code. This refinement is based on models for the runtime of different code regions.

Future extensions to SimAnMo include the automation of the whole model creation process. In particular, we stress on the selection of the training data points. Currently, manual inspection is required to remove heavy outliers or to detect ranges of p where the program behaves differently from high p-values. Furthermore, the numerical stability and the performance of the *MAPE*-approach to determine c_0 and c_1 will be increased. Either by tuning the parameters to configure the conjugate gradient process or by switching to another minimization procedure.

ACKNOWLEDGMENTS

We thank the participants and organizers of the PMBS2020 workshop for their interesting questions concerning SimAnMo which motivated much of the research and the results presented here. We thank the anonymous reviewers of this article for their valuable feedback. We also thank Y. Wang for the fruitful discussions about the pBKZ library and that he granted us access to the measurement data. Our work is partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—SFB 1119 236615297. Calculations for this research were conducted on the Lichtenberg high performance computer of the TU Darmstadt.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in [SimAnMo] at [https://github.com/tudasc/SimAnMo]

ENDNOTES

*https://github.com/tudasc/SimAnMo
†https://www.latticechallenge.org/svp-challenge/
‡https://www.latticechallenge.org/svp-challenge/download/generator.zip
§https://github.com/Qumeric/tsp-suboptimal

ORCID

Michael Burger https://orcid.org/0000-0002-1462-2543
Christian Bischof https://orcid.org/0000-0003-2711-3032

REFERENCES

1. Costello C, Longa P, Naehrig M, Renes J, Virdia F. Improved classical cryptanalysis of sike in practice. cryptology ePrint archive, report 2019/298; 2019. https://eprint.iacr.org/2019/298
2. Wunderer T, Burger M, Nguyen GN. Parallelizing the hybrid lattice-reduction and meet-in-the-middle attack. In: Pop F, Negru C, Gonzalez-Velez H, Rak J, eds. *Proceedings of the 2018 IEEE International Conference on Computational Science and Engineering (CSE)*. IEEE; 2018:185-193.
3. Niewiadomska-Szynkiewicz E, Marks M, Jantura J, Podbielski M, Strzelczyk P. Comparative study of massively parallel cryptalysis and cryptography on CPU-GPU cluster. Proceedings of the 2013 Military Communications and Information Systems Conference; 2013:1-8.
4. Ducas L. Shortest vector from lattice sieving: a few dimensions for free. In: Nielsen JB, Rijmen V, eds. *Advances in Cryptology – EUROCRYPT 2018*. Springer International Publishing; 2018:125-145.
5. Albrecht MR, Ducas L, Herold G, Kirshanova E, Postlethwaite EW, Stevens M. The general sieve kernel and new records in lattice reduction. cryptology ePrint archive, report 2019/089; 2019. https://eprint.iacr.org/2019/089

6. Aono Y, Wang Y, Hayashi T, Takagi T. Improved progressive BKZ algorithms and their precise cost estimation by sharp simulator. In: Fischlin M, Coron J, eds. *Advances in Cryptology - EUROCRYPT Proceedings of the 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I. 9665 of Lecture Notes in Computer Science*. Vol 2016. Springer; 2016:789-819.
7. Hoffstein J, Howgrave-Graham N, Pipher J, Whyte W. *Practical Lattice-Based Cryptography: NTRUEncrypt and NTRUSign*. Springer; 2009:349-390.
8. Peikert C. Lattice cryptography for the internet. In: Mosca M, ed. *Post-Quantum Cryptography*. Springer International Publishing; 2014:197-219.
9. Gama N, Nguyen PQ, Regev O. Lattice enumeration using extreme pruning. In: Gilbert H, ed. *Advances in Cryptology – EUROCRYPT*. Springer; 2010:257-278.
10. Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures. *Commun ACM*. 2009;52(4):65-76.
11. Ilic A, Pratas F, Sousa L. Cache-aware roofline model: upgrading the loft. *IEEE Comput Archit Lett*. 2013;13(1):21-24.
12. Denoyelle N, Goglin B, Ilic A, Jeannot E, Sousa L. Modeling non-uniform memory access on large compute nodes with the cache-aware roofline model. *IEEE Trans Parallel Distrib Syst*. 2019;30(6):1374-1389.
13. Lopes A, Pratas F, Sousa L, Ilic A. Exploring GPU performance, power and energy-efficiency bounds with cache-aware roofline modeling. In: Papamichael MK, ed. *Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE; 2017:259-268.
14. Ding N, Williams S. An instruction roofline model for GPUs. In: Wright SA, Jarvis SA, Hammond SD, eds. *Proceedings of the 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE; 2019:7-18.
15. Lorenzo OG, Pena TF, Cabaleiro JC, Pichel JC, Rivera FF. 3DyRM: a dynamic roofline model including memory latency information. *J Supercomput*. 2014;70(2):696-708.
16. Malakar P, Balaprakash P, Vishwanath V, Morozov V, Kumaran K. Benchmarking machine learning methods for performance modeling of scientific applications. In: Wright SA, Jarvis SA, Hammond SD, eds. *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE; 2018:33-44.
17. Ritter M, Calotoiu A, Rinke S, Reimann T, Hoefler T, Wolf F. Learning cost-effective sampling strategies for empirical performance modeling. In: Chard K, ed. *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE; 2020:884-895.
18. Calotoiu A, Copik M, Hoefler T, Ritter M, Shudler S, Wolf F. ExtraPeak: advanced automatic performance modeling for HPC applications. In: Bungartz HJ, Reiz S, Uekermann B, Neumann P, Nagel WE, eds. *Software for Exascale Computing - SPPEXA 2016-2019*. Springer International Publishing; 2020:453-482.
19. Burger M, Bischof C, Calotoiu A, Wunderer T, Wolf F. Exploring the performance envelope of the LLL algorithm. In: Pop F, Negru C, Gonzalez-Velez H, Rak J, eds. *Proceedings of the 2018 IEEE International Conference on Computational Science and Engineering (CSE)*, IEEE; 2018:36-43.
20. Calotoiu A, Hoefler T, Poke M, Wolf F. Using automated performance modeling to find scalability bugs in complex codes. Proceedings of the ACM/IEEE Conference on Supercomputing (SC13); 2013:1-12; Denver, CO.
21. Ilyas MK, Calotoiu A, Wolf F. *Off-Road Performance Modeling – How to Deal with Segmented Data*. Springer International Publishing; 2017:36-48.
22. Hyndman RJ, Koehler AB. Another look at measures of forecast accuracy. *Int J Forecast*. 2006;22(4):679-688. doi:10.1016/j.ijforecast.2006.03.001
23. Reisert P, Calotoiu A, Shudler S, Wolf F. Following the blind seer – creating better performance models using less information. Proceedings of the European Conference on Parallel Processing Lecture Notes in Computer Science; 2017:106-118; Springer
24. Burger M, Nguyen GN, Bischof C. Developing models for the runtime of programs with exponential runtime behavior. In: Wright SA, Jarvis SA, Hammond SD, eds. *Proceedings of the 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE; 2020:109-125.
25. Guennebaud G, Jacob B. Eigen v3; 2010. http://eigen.tuxfamily.org
26. ALGLIB Project. ALGLIB - numerical analysis library; 1999. https://www.alglib.net/
27. Grover LK. From Schrödinger's equation to the quantum search algorithm. *Pramana*. 2001;56(2-3):333-348.
28. Khot S. Hardness of approximating the shortest vector problem in lattices. *J ACM*. 2005;52(5):789-808.
29. Bernstein DJ, Buchmann J, Dahmen E. *Post Quantum Cryptography*. 1st ed. Springer Publishing Company; 2008.
30. Becker A, Ducas L, Gama N, Laarhoven T. New directions in nearest neighbor searching with applications to lattice sieving. In: Kraughgamer R, ed. *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms SODA '16*. Society for Industrial and Applied Mathematics; 2016:10-24.
31. Burger M, Bischof CH, Krämer J. p3Enum: a new parameterizable and shared-memory parallelized shortest vector problem solver. In: Rodrigues JMF, Cardoso PJS, Monteiro JM, et al., eds. *Computational Science - ICCS 2019 - 19th International Conference, Faro, Portugal, June 12-14, 2019, Proceedings, Part V. 11540 of Lecture Notes in Computer Science*. Springer; 2019:535-542.
32. Nguyen GN. *Developing Performance Models for Codes with Exponential Runtime Behavior*. Master's thesis. Technische Universität Darmstadt; 2019. https://tubiblio.ulb.tu-darmstadt.de/118493/
33. Rivest R, Shamir A, Adleman L. A method for obtaining digital signatures and public-key cryptosystems. *Commun ACM*. 1978;21:120-126.
34. Lehr JP, Hück A, Bischof C. PIRA: performance instrumentation refinement automation. *AI-SEPS 2018*. ACM; 2018:1-10.

How to cite this article: Burger M, Nguyen GN, Bischof C. SimAnMo—A parallelized runtime model generator. *Concurrency Computat Pract Exper*. 2022;34(20):e6771. doi: 10.1002/cpe.6771